gitk is a graphical history viewer. Think of it like a powerful GUI shell over git log and git grep

Gitk is easiest to invoke from the command-line. Just cd into a Git repository, and type:

$ gitk [git log options]

Gitk accepts many command-line options, most of which are passed through to the underlying git log action. Probably one of the most useful is the --all flag, which tells gitk to show commits reachable from *any* ref, not just HEAD.

git-gui, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

$ git gui

Pro Git (Vf)

$ git log –abrev-commit –pretty=oneline

Pour connaitre l’empreinte SHA sur laquelle pointe une branche

$ git rev-parse topic1

Git maintient en arriere-plan un historique des references ou sont passes HEAD et vos branches sur les derniers mois — ceci s’appelle le *reflog*. Vous pouvez le consulter avec la commande git reflog :

Si vous souhaitez consulter le n-ieme antecedent de votre HEAD, vous pouvez utiliser la reference @{n} du reflog, 5 dans cet exemple :

$ git show HEAD@{5}

Pour consulter le reflog au format git log, executez: git log -g. Veuillez noter que le reflog ne stocke que des informations locales, c’est un historique de ce que

vous avez fait dans votre depot

Alors, vous pouvez consulter le *commit* precedent en specifiant HEAD^, ce qui signifie ≪ le parent de HEAD ≫:

Vous pouvez egalement specifier un nombre apres ^ — par exemple, d921970^2 signifie ≪ le second parent de d921970 ≫. Cette syntaxe ne sert que pour les *commits* de fusion, qui ont plus d’un parent. Le premier parent est la branche depuis laquelle vous avez fusionne, et le second est le *commit* de la branche que vous avez fusionnee :

Une autre solution courante pour specifier une reference ancetre est le ~. Il fait egalement reference au premier parent, donc HEAD~ et HEAD^ sont equivalents. La difference apparait si vous specifiez un nombre. HEAD~2 signifie ≪ le premier parent du premier parent ≫, ou bien ≪ le grandparent ≫ ; on remonte les premiers parents autant de fois que demande.

HEAD~3 peut aussi s’ecrire HEAD^^^.

Vous pouvez egalement combiner ces syntaxes HEAD~3^2,

**Plages de** *commits*

**Double point**

Si vous voulez savoir ce qui n’a pas encore ete fusionne sur votre branche master depuis votre branche experiment, vous pouvez demander a Git de vous montrer un journal des *commits* avec master..experiment

$ git log origin/master..HEAD

Cette commande vous affiche tous les *commits* de votre branche courante qui ne sont pas sur la branche master du depot distant origin.

Vous pouvez egalement laisser tomber une borne de la syntaxe pour faire comprendre a Git que vous parlez de HEAD. Par exemple, vous pouvez obtenir les memes resultats que precedemment en tapant git log origin/master.. — Git utilise HEAD si une des bornes est manquante.

**Emplacements multiples**

Git vous permet cela avec ^ ou --not en prefixe de toute reference de laquelle vous ne souhaitez pas voir les *commits*. Les 3 commandes ci-apres sont equivalentes :

$ git log refA..refB

$ git log ^refA refB

$ git log refB --not refA

C’est utile car cela vous permet de specifier plus de 2 references dans votre requete, ce que vous ne pouvez accomplir avec la syntaxe double-point. Par exemple, si vous souhaitez voir les *commits* qui sont accessibles depuis refA et refB mais pas depuis refC, vous pouvez taper ces 2 commandes :

$ git log refA refB ^refC

$ git log refA refB --not refC

**Triple point**

La derniere syntaxe majeure de selection de plage de *commits* est la syntaxe triple-point qui specifie tous les *commits* accessibles par l’une des deux references, exclusivement

$ git log master...experiment

Une option courante a utiliser avec la commande log dans ce cas est --left-right qui vous montre la borne de la plage a laquelle ce *commit* appartient. Cela rend les donnees plus utiles :

$ git log --left-right master...experiment

Mode patch

$ git add –p

$ git reset –p

$ git checkout –p

$ git stash save -p

Remiser votre travail

$ git stash save

$ git stash list

$ git stash apply

vous voulez appliquer une remise plus ancienne, vous pouvez la specifier en la nommant, comme ceci : git stash apply stash@{2}. Si vous ne specifiez pas une remise, Git presume que vous voulez la remise la plus recente et essaye de l’appliquer

Par defaut, les modifications de vos fichiers sont re-appliquees, mais pas les indexations. Pour cela, vous devez executer la commande git stash apply avec l’option --index pour demander a Git d’essayer de re-appliquer les modifications de votre index.

Pour la supprimer, vous pouvez executer git stash drop avec le nom de la remise a supprimer

Vous pouvez egalement executer git stash pop pour appliquer et supprimer immediatement la remise de votre pile.

Il existe des variantes de remisages qui peuvent s’averer utiles.La premiere option

est l’option --keep-index de la commande stash save. Elle indique a Git de ne pas remiser ce qui aurait ete deja indexe au moyen de la commande git add.

Une autre option utile de stash est la possibilite de remiser les fichiers non suivis aussi bien que les fichiers suivis. Par defaut, git stash ne sauve que les fichiers qui sont deja suivis ou indexes. Si vous specifiez l’option --include-untracked ou -u, Git remisera aussi les fichiers non-suivis du repertoire de travail

Une option plus securisee consiste a lancer git stash --all pour tout sauvegarder dans une remise.

**Creer une branche depuis une remise**

vous pouvez executer git stash branch qui creera une nouvelle branche a votre place, recuperant le *commit* ou vous etiez lorsque vous avez cree la remise, re-appliquera votre travail dedans, et supprimera finalement votre remise si cela a reussi

$ git stash branch testchanges

**Nettoyer son repertoire de travail**

Pour supprimer tous les fichiers non-suivis, vous pouvez lancer git clean -f -d, qui effacera aussi tout sous-repertoire vide. L’option -f signifie ≪ force ≫, soit ≪ fais-le reellement ≫.

$ git clean -d -n

Si vous souhaitez visualiser ce qui serait fait, vous pouvez lancer la commande avec l’option -n qui signifie ≪ fais-le a blanc et montre-moi ce qui *serait* supprime ≫.

$ git clean -d –n

Par defaut, la commande git clean ne va supprimer que les fichiers non-suivis qui ne sont pas ignores. Tout fichier qui correspond a un motif de votre fichier .gitignore ou tout autre fichier similaire ne sera pas supprime. Si vous souhaitez supprimer aussi ces fichiers, comme par exemple les fichiers .o genere par un compilateur pour faire une compilation totale, vous pouvez ajouter l’option -x a la commande de nettoyage

Si vous ne savez pas ce que la commande git clean va effectivement supprimer, lancez-la une premiere fois avec -n par securite avant de transformer le -n en -f et nettoyer definitivement. Un autre choix pour s’assurer de ce qui va etre efface consiste a lancer la commande avec l’option –i ou --interactive.

La commande sera lancee en mode interactif.

$ git clean -x –i

**Recherche**

Git fournit quelques outils permettant rapidement de rechercher dans le code et les *commits* stockes dans votre base de données

**Git grep**

grep permet de rechercher facilement une chaine de caracteres ou une expression reguliere dans une arborescence validee ou dans le repertoire de travail. Pour tous les exemples qui suivent, nous allons utiliser le depot de Git lui-meme. Par defaut, git grep recherche dans le repertoire de travail. Vous pouvez passer l’option -n pour

afficher les numeros des lignes des correspondances.

$ git grep -n gmtime\_r

La commande grep peut etre enrichie grace a un certain nombre d’options nteressantes.

vous pouvez indiquer a Git de resumer le resultat en ne montrant que les fichiers et le nombre de correspondances au moyen de l’option --count :

$ git grep --count gmtime\_r

Si vous souhaitez voir dans quelle methode ou fonction la correspondance a ete trouvee, vous pouvez passer l’option -p :

$ git grep -p gmtime\_r \*.c

Vous pouvez aussi rechercher des combinaisons plus complexes de chaines de caracteres avec l’option --and

Ici, nous allons utiliser les options --break et --heading qui aident a decouper le resultat dans un format plus digeste.

Par exemple, recherchons toutes les lignes qui definissent une constante qui contient au choix ≪ LINK ≫ ou ≪ BUF\_MAX ≫ dans la base de code de Git avant la version 1.8.0.

$ git grep --break --heading -n -e '#define' --and ( -e LINK -e BUF\_MAX ) v1.8.0

La commande git grep a quelques avantages sur les commandes de recherche normales telles que grep et ack. Le premier est qu’elle est vraiment rapide, le second est qu’elle vous permet de rechercher dans n’importe quelle arborescence Git, pas seulement la copie de travail. Comme nous l’avons vu dans l’exemple ci-dessus, nous avons cherche des termes dans une version ancienne du code source de Git, pas dans la derniere version extraite.

**Recherche dans le journal Git**

Peut-etre ne cherchez-vous pas **ou** un terme apparait, mais plutot **quand** il existait ou fut introduit. La commande git log comprend un certain nombre d’outils puissants pour trouver des *commits* specifiques par le contenu de leurs messages ou le contenu des diffs qu’ils introduisent.

Si vous voulez trouver par exemple quand la constante ZLIB\_BUF\_MAX a ete initialement introduite, nous pouvons indiquez a Git de ne montrer que les *commits* qui soit ajoutent soit retirent la chaine avec l’option -S.

$ git log -SZLIB\_BUF\_MAX –oneline

Si vous devez etre plus specifique, vous pouvez fournir une expression reguliere a rechercher avec l’option -G.

**Recherche des evolutions d’une ligne**

On l’active avec l’option -L de git log et elle vous montre l’historique d’une fonction ou d’une ligne dans votre base de code.

Par exemple, si nous souhaitions voir toutes les modifications realisees sur la fonction git\_deflate\_bound dans le fichier zlib.c, nous pourrions lancer git log -L

:git\_deflate\_bound:zlib.c. Cette commande va essayer de determiner les limites de cette fonction et de rechercher dans l’historique chaque modification realisee sur la fonction comme une serie de patchs jusqu’au moment de sa creation.

$ git log -L :git\_deflate\_bound:zlib.c

i Git ne peut pas determiner comment trouver la fonction ou la methode dans votre langage de programmation, vous pouvez aussi fournir une regex. Par exemple, cela aurait donne le meme resultat avec git log -L '/unsigned long git\_deflate\_bound/',/^}/:zlib.c. Vous auriez pu aussi specifier un intervalle de lignes ou un numero de ligne et vous auriez obtenu le meme type de resultat.

**Reecrire l’historique**

**Modifier la derniere validation**

$ git commit –amend

Vous indexez les modifications que vous voulez en executant git add ou git rm, et le prochain git commit –amend prendra votre index courant et en fera le contenu de votre nouvelle validation.

**Modifier plusieurs messages de validation**

**Reordonner les** *commits*

**Ecraser un** *commit*

**Diviser un** *commit*

Par exemple, si vous voulez modifier les 3 derniers messages de validation ou n’importe lequel des messages dans ce groupe, vous fournissez a git rebase -i le parent du dernier *commit* que vous voulez editer,

$ git rebase -i HEAD~3

Souvenez-vous egalement que ceci est une commande de rebasage, chaque *commit* inclus dans l intervalle HEAD~3..HEAD sera reecrit,

**Reset demystifie**

Le moyen le plus simple de penser a reset et checkout consiste a representer Git comme un gestionnaire de contenu de trois arborescences differentes.

Git, comme systeme, gere et manipule trois arbres au cours de son operation normale :

**Arbre Role**

HEAD instantane de la derniere validation, prochain parent

Index instantane propose de la prochaine validation

Repertoire de travail bac a sable

**HEAD**

HEAD est un pointeur sur la reference de la branche actuelle, qui est a son tour un pointeur sur le dernier *commit* realise sur cette branche. Ceci signifie que HEAD sera le parent du prochain *commit* a creer.

$ git cat-file -p HEAD

$ git ls-tree -r HEAD

**L’index**

L’index est votre **prochain commit propose**. Nous avons aussi fait reference a ce concept comme la ≪ zone de preparation ≫ de Git du fait que c’est ce que Git examine lorsque vous lancez git commit.

$ git ls-files –s

Encore une fois, nous utilisons ici ls-files qui est plus une commande de coulisses qui vous montre l’etat actuel de votre index. L’index n’est pas techniquement parlant une structure arborescente ‑ c’est en fait un manifeste aplati ‑ mais pour nos besoins, c’est suffisamment proche.

**Le repertoire de travail**

Il faut penser a la copie de travail comme un **bac a sable** ou vous pouvez essayer vos modifications avant de les transferer dans votre index puis le valider dans votre historique.

Reset manipule directement les trois arborescences d’une manière simple et prédictible. Il réalise jusqu’à trois opérations basiques.

**Etape 1: deplacer HEAD**

La premiere chose que reset va faire consiste a deplacer ce qui est pointe par HEAD. Ce n’est pas la meme chose que changer HEAD lui-meme (ce que fait checkout). reset deplace la branche que HEAD pointe.

Quelle que soit la forme du reset que vous invoquez pour un *commit*, ce sera toujours la premiere chose qu’il tentera de faire. Avec reset --soft, il n’ira pas plus loin.

**Etape 2 : Mise a jour de l’index (--mixed)**

**Etape 3: Mise a jour de la copie de travail (--hard)**

La troisième chose que reset va faire est de faire correspondre la copie de travail avec l’index. Si vous utilisez l’option --hard, il continuera avec cette etape.

Il est important de noter que cette option (--hard) est le seul moyen de rendre la commande reset dangereuse et est un des tres rares cas ou Git va reellement detruire de la donnee. Toute autre invocation de reset peut etre defaite,

**Recapitulatif**

La commande reset remplace ces trois arbres dans un ordre specifique, s’arretant lorsque vous lui indiquez :

1. Deplace la branche pointee par HEAD *(s’arrete ici si* --soft*)*

2. Fait ressembler l’index a HEAD *(s’arrete ici a moins que* --hard*)*

3. Fait ressembler le repertoire de travail a l’index.

**Reset avec un chemin**

Donc, supposons que vous lancez git reset file.txt. Cette forme (puisque vous n’avez pas specifie un SHA-1 de commit ni de branche, et que vous n’avez pas non plus specifie --soft ou --hard) est un raccourcis pour git reset --mixed HEAD file.txt.

il ne fait que copier file.txt de HEAD vers index

Ceci a l’effet pratique de *desindexer* le fichier. Si on regarde cette commande dans le diagramme et qu’on pense a ce que git add fait, ce sont des opposes exacts

Nous pourrions tout aussi bien ne pas laisser Git considerer que nous voulions dire ≪ tirer les donnees depuis HEAD ≫ en specifiant un *commit* specifique d’ou tirer ce fichier. Nous lancerions juste quelque chose comme git reset eb43bf file.txt.

**Et checkout**

Comme reset, checkout manipule les trois arborescences et se comporte generalement differemment selon que vous indiquez un chemin vers un fichier ou non.

**Sans chemin**

Lancer git checkout [branche] est assez similaire a lancer git reset --hard [branche] en ce qu’il met a jour les trois arborescences pour qu’elles ressemblent a [branche], mais avec deux differences majeures

Premierement, a la difference de reset --hard, checkout preserve le repertoire de travail ; il s’assure de ne pas casser des fichiers qui ont change. En fait, il est meme un peu plus intelligent que ca – il essaie de faire une fusion simple dans le repertoire de travail, de facon que tous les fichiers **non modifies** soient mis a jour. reset --hard, par contre, va simplement tout remplacer unilateralement sans rien verifier.

Donc, dans les deux cas, nous deplacons HEAD pour pointer sur le commit A, mais la manière differe beaucoup. reset va deplacer la branche pointee par HEAD, alors que checkout va deplacer HEAD lui-meme.



**Avec des chemins**

L’autre facon de lancer checkout est avec un chemin de fichier, ce qui, comme reset, ne deplace pas HEAD. Cela correspond juste a git reset [branche] fichier car cela met a jour l’index avec ce fichier a ce *commit*, mais en remplacant le fichier dans le repertoire de travail. Ce serait exactement comme git reset --hard [branche] fichier (si reset le permettait) – cela ne preserve pas le repertoire de travail et ne deplace pas non plus HEAD.

**Resume**

Voici un aide-memoire sur ce que chaque commande affecte dans chaque arborescence. La colonne ≪ HEAD ≫ contient ≪ REF ≫ si cette commande deplace la reference (branche) pointee par HEAD, et ≪ HEAD ≫ si elle deplace HEAD lui-meme. Faites particulierement attention a la colonne ≪ preserve RT ? ≫ (preserve le repertoire de travail) – si elle indique **NON**, reflechissez a deux fois avant de lancer la commande.



**Fusion avancee**

**Conflits de fusion**

Premierement, nous entrons dans l’etat de conflit de fusion. Puis, nous voulons obtenir des copies de la version locale (**ours**), de la version distante (**theirs**, celle qui vient de la branche a fusionner) et de la version commune (l’ancetre commun depuis lequel les branches sont parties). Ensuite, nous voulons corriger au choix la version locale ou la distante et reessayer de fusionner juste ce fichier.

Obtenir les trois versions des fichiers est en fait assez facile. Git stocke toutes ces versions dans l’index sous formes d’etapes (**stages**), auxquelles chacune y a un nombre associe. Stage 1 est l’ancetre commun, stage 2 est notre version, stage 3 est la version de MERGE\_HEAD, la version qu’on cherche a fusionner (**theirs**).

Vous pouvez extraire une copie de chacune de ces versions du fichier en conflit avec la commande git show et une syntaxe speciale.

$ git show :1:hello.rb > hello.common.rb

$ git show :2:hello.rb > hello.ours.rb

$ git show :3:hello.rb > hello.theirs.rb

Si vous voulez rentrer un peu plus dans le dur, vous pouvez aussi utiliser la commande de plomberie ls-files -u pour recuperer les SHA-1 des blobs Git de chacun de ces fichiers.

$ git ls-files –u

La syntaxe :1:hello.rb est juste un raccourcis pour la recherche du SHA-1 de ce blob.

A present que nous avons le contenu des trois etapes dans notre repertoire de travail, nous pouvons reparer manuellement la copie distante pour resoudre le probleme d’espaces et refusionner le fichier avec la commande meconnue git merge-file dont c’est l’exacte fonction

$ git merge-file -p hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

vous pouvez demander a git diff de comparer le contenu de votre repertoire de travail que vous etes sur le point de valider comme resultat de la fusion avec n’importe quelle etape. Detaillons chaque comparaison. Pour comparer votre resultat avec ce que vous aviez dans votre branche avant la fusion, en d’autres termes, ce que la fusion a introduit, vous pouvez lancer git diff –ours

Si nous voulons voir le resultat de la fusion modifiee depuis la version distante, nous pouvons lancer git diff --theirs.

Enfin, nous pouvons voir comment le fichier a ete modifie dans les deux branches avec git diff --base.

**Examiner les conflits**

Si nous ouvrons le fichier, nous verrons quelque chose comme :

#! /usr/bin/env ruby

def hello

<<<<<<< HEAD

puts 'hola world'

======

puts 'hello mundo'

>>>>>>> mundo

end

hello()

Peut-etre le moyen de resoudre n’est-il pas evident. Il necessite plus de contexte.

Un outil utile est git checkout avec l’option --conflict. Il va re-extraire le fichier et remplacer les marqueurs de conflit. Cela peut etre utile si vous souhaitez eliminer les marqueurs et essayer de resoudre le conflit a nouveau.

Vous pouvez passer en parametre a --conflict, soit diff3 soit merge (le parametre par defaut). Si vous lui passez diff3, Git utilisera une version differente des marqueurs de conflit, vous fournissant non seulement les versions locales (*ours*) et distantes (*theirs*), mais aussi la version ≪ base ≫ integree pour vous fournir plus de contexte.

$ git checkout --conflict=diff3 hello.rb

Une fois que nous l’avons lance, le fichier ressemble a ceci :

#! /usr/bin/env ruby

def hello

<<<<<<< ours

puts 'hola world'

||||||| base

puts 'hello world'

======

puts 'hello mundo'

>>>>>>> theirs

end

hello()

Si vous appreciez ce format, vous pouvez le regler comme defaut pour les futur conflits de fusion en renseignant le parametre merge.conflictstyle avec diff3.

$ git config --global merge.conflictstyle diff3

La commande git checkout peut aussi accepter les options --ours et --theirs, qui peuvent servir de moyen rapide de choisir unilateralement une version ou une autre sans fusion

Cela peut etre particulierement utile pour les conflits de fichiers binaires ou vous ne pouvez que choisir un des cote, ou des conflits ou vous souhaitez fusionner certains fichiers depuis d’autres branches - vous pouvez fusionner, puis extraire certains fichiers depuis un cote ou un autre avant de valider le resultat.

**Journal de fusion**

Un autre outil utile pour la resolution de conflits de fusion est git log. Cela peut vous aider a obtenir du contexte ce qui a contribue aux conflits. Parcourir un petit morceau de l’historique pour se rappeler pourquoi deux lignes de developpement ont touche au meme endroit dans le code peut s’averer quelque fois tres utile.

Pour obtenir une liste complete de tous les *commits* uniques qui ont ete introduits dans chaque branche impliquee dans la fusion, nous pouvons utiliser la syntaxe ≪ triple point ≫ que nous avons apprise dans Triple point.

$ git log --oneline --left-right HEAD...MERGE\_HEAD

Neanmoins, nous pouvons simplifier encore plus ceci pour fournir beaucoup plus de contexte. Si nous ajoutons l’option --merge a git log, il n’affichera que les *commits* de part et d’autre de la fusion qui modifient un fichier presentant un conflit.

$ git log --oneline --left-right –merge

Si nous lancons cela avec l’option -p a la place, vous obtenez les diffs limites au fichier qui s’est retrouve en conflit. Cela peut s’averer **vraiment** utile pour vous donner le contexte necessaire a la comprehension de la raison d’un conflit et a sa resolution intelligente.

**Defaire des fusions**

**Correction des references**

git reset --hard HEAD~

Le defaut de cette approche est qu’elle re-ecrit l’historique, ce qui peut etre problematique avec un depot partage.

**Inverser le** *commit*

Si les deplacements des pointeurs de branche ne sont pas envisageables, Git vous donne encore l’option de creer un nouveau *commit* qui defait toutes les modifications d’un autre deja existant. Git appelle cette option une ≪ inversion ≫ (*revert*), et dans ce scenario particulier, vous l’invoqueriez comme ceci :

$ git revert -m 1 HEAD

**Autres types de fusions**

**Preference** *our* **ou** *theirs*

Premierement, il existe un autre mode utile que nous pouvons utiliser avec le mode ≪ recursive ≫ normal de fusion. Nous avons deja vu les options ignore-all-space et ignore-space-change qui sont passees avec -X mais nous pouvons aussi indiquer a Git de favoriser un cote plutot que l’autre lorsqu’il rencontre un conflit.

Si vous preferez que Git choisisse simplement un cote specifique et qu’il ignore l’autre cote au lieu de vous laisser fusionner manuellement le conflit, vous pouvez passer -Xours ou -Xtheirs a la commande merge.

Si une des options est specifiee, Git ne va pas ajouter de marqueurs de conflit. Toutes les differences qui peuvent etre fusionnees seront fusionnees. Pour toutes les differences qui generent un conflit, Git choisira simplement la version du cote que vous avez specifie, y compris pour les fichiers binaires.

$ git merge -Xours mundo

Cette option peut aussi etre passee a la commande git merge-file que nous avons utilisee plus tot en lancant quelque chose comme git merge-file --ours pour les fusions de fichiers individuels

Si vous voulez faire quelque chose similaire mais indiquer a Git de ne meme pas essayer de fusionner les modifications de l’autre cote, il existe une option draconienne qui s’appelle la *strategie* de fusion ≪ *ours* ≫.

Cela realisera une fusion factice. Cela enregistrera un nouveau *commit* de fusion avec les deux branches comme parents, mais ne regardera meme pas la branche en cours de fusion. Cela enregistrera simplement le code exact de la branche courante comme resultat de la fusion.

$ git merge -s ours mundo

Vous pouvez voir qu’il n’y a pas de difference entre la branche sur laquelle nous etions precedemment et le resultat de la fusion.

**Deboguer avec Git**

vous pouvez visualiser le fichier annote avec git blame pour voir quand chaque ligne de la methode a ete modifiee pour la derniere fois et par qui. Cet exemple utilise l’option -L pour limiter la sortie des lignes 12 a 22 :

$ git blame -L 12,22 simplegit.rb

**Sous-modules**

**Demarrer un sous-module**

Pour ajouter un nouveau sous-module, nous utilisons la commande git submodule add avec l’URL du projet que nous souhaitons suivre

$ git submodule add https://github.com/chaconinc/DbConnector

Si vous lancez git status a ce moment, vous noterez quelques differences.

Premierement, un fichier .gitmodules vient d’apparaitre. C’est le fichier de configuration qui stocke la liaison entre l’URL du projet et le sous-repertoire local dans lequel vous l’avez tire.

$ cat .gitmodules

[submodule "DbConnector"]

path = DbConnector

url = <https://github.com/chaconinc/DbConnector>

fichier. Il est important de noter que ce fichier est en gestion de version comme vos autres fichiers, a l’instar de votre fichier .gitignore. Il est pousse et tire comme le reste de votre projet.

$ git diff DbConnector

Meme si DbConnector est un sous-repertoire de votre repertoire de travail, Git le voit comme un sous-module et ne suit pas son contenu (si vous n’etes pas dans ce repertoire). En echange, Git l’enregistre comme un *commit* particulier de ce depot. Si vous souhaitez une sortie diff plus agreable, vous pouvez passer l’option --submodule a git diff.

$ git diff –submodule

**Cloner un projet avec des sous-modules**

Maintenant, vous allez apprendre a cloner un projet contenant des sous-modules. Quand vous recuperez un tel projet, vous obtenez les differents repertoires qui contiennent les sous-modules, mais encore aucun des fichiers

Le repertoire DbConnector est present mais vide. Vous devez executer deux commandes : git submodule init pour initialiser votre fichier local de configuration, et git submodule update pour tirer toutes les donnees de ce projet et recuperer le *commit* approprie tel que liste dans votre superprojet:

$ git submodule init

$ git submodule update

Il existe une autre maniere plus simple d’arriver au meme resultat. Si vous passez l’option --recurse-submodules a la commande git clone, celle-ci initialisera et mettra a jour automatiquement chaque sous-module du depot.

**Travailler sur un projet comprenant des sous-modules**

**Tirer des modifications amont**

Quand vous souhaitez vérifier si le sous-module a évolue, vous pouvez vous rendre dans le répertoire correspondant et lancer git fetch puis git merge de la branche amont pour mettre a jour votre code local.

$ git fetch

$ git merge origin/master

Si vous revenez maintenant dans le projet principal et lancez git diff --submodule, vous pouvez remarquer que le sous-module a ete mis a jour et vous pouvez obtenir une liste des *commits* qui y ont ete ajoutes

Il existe aussi un moyen plus facile, si vous preferez ne pas avoir a recuperer et fusionner manuellement les modifications dans le sous-repertoire. Si vous lancez la commande git submodule update --remote, Git se rendra dans vos sous-modules et realisera automatiquement le fetch et le merge.

C’est une information interessante car vous pouvez voir le journal des modifications que vous vous appretez a valider dans votre sous-module. Une fois validees, vous pouvez encore visualiser cette information en lancant git log -p.

$ git log -p –submodule

Par defaut, Git essaiera de mettre a jour **tous** les sous-modules lors d’une commande git submodule update --remote, donc si vous avez de nombreux sous-modules, il est preferable de specifier le sousmodule que vous souhaitez mettre a jour.

$ git submodule update --remote DbConnector

**Travailler sur un sous-module**

En premier, rendons-nous dans le repertoire de notre sous-module et extrayons une branche.

$ git checkout stable

Attaquons-nous au choix de politique de gestion. Pour le specifier manuellement, nous pouvons simplement ajouter l’option --merge or --rebasea l’appel de update.

$ git submodule update --remote –merge

Maintenant, si nous mettons a jour notre sous-module, nous pouvons voir ce qui arrive lors d’un rebasage de deux modifications concurrentes.

$ git submodule update --remote –rebase

**Publier les modifications dans un sous-module**

Pour etre sur que cela n’arrive pas, vous pouvez demander a Git de verifier que tous vos sousmodules ont ete correctement pousses avant de pouvoir pousser le projet principal. La commande git push accepte un argument --recurse-submodules qui peut avoir pour valeur ≪ *check* ≫ ou ≪ *on-demand* ≫. L’option ≪ check ≫ fera echouer push si au moins une des modifications des sous-modules n’a pas ete poussee.

Comme vous pouvez le voir, Git s’est rendu dans le module DbConnector et l’a pousse avant de pousser le projet principal. Si la poussee du sous-module echoue pour une raison quelconque, la poussee du projet principal sera annulee.

**Trucs et astuces pour les sous-modules**

Il existe quelques commandes qui permettent de travailler plus facilement avec les sous-modules.

**Submodule foreach**

Il existe une commande submodule foreach qui permet de lancer une commande arbitraire dans chaque sous-module. C’est particulierement utile si vous avez plusieurs sous-modules dans le meme projet

$ git submodule foreach git stash

Ensuite, nous pouvons creer une nouvelle branche et y basculer dans tous nos sous-modules.

$ git submodule foreach 'git checkout -b featureA'

$ git diff; git submodule foreach 'git diff'

Les alias Git, mais voici un exemple d’alias que vous pourriez trouver utiles si vous voulez travailler serieusement avec les sous-modules de Git.

$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"

$ git config alias.spush 'push --recurse-submodules=on-demand'

$ git config alias.supdate 'submodule update --remote --merge'

**Configuration de Git**

**Configuration de base d’un client**

core.editor:

Pour modifier ce programme par defaut pour un autre, vous pouvez utiliser le parametre core.editor

$ git config --global core.editor emacs

commit.template:

Si vous reglez ceci sur le chemin d’un fichier sur votre systeme, Git utilisera ce fichier comme message par defaut quand vous validez

$ git config --global commit.template ~/.gitmessage.txt

core.autocrlf

$ git config --global core.autocrlf true

$ git config --global core.autocrlf input

Si vous etes un programmeur Windows gerant un projet specifique a Windows, vous pouvez desactiver cette fonctionnalite et forcer l’enregistrement des ≪ retour chariot ≫ dans le depot en reglant la valeur du parametre a false :

$ git config --global core.autocrlf false

**Plomberie et porcelaine**

Quand vous executez git init dans un nouveau repertoire ou un repertoire existant, Git cree un repertoire .git qui contient presque tout ce que Git stocke et manipule.

$ ll -la .git

total 11

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 ./

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 ../

-rw-r--r-- 1 asaki 1049089 130 Sep 27 18:37 config

-rw-r--r-- 1 asaki 1049089 73 Sep 27 18:37 description

-rw-r--r-- 1 asaki 1049089 23 Sep 27 18:37 HEAD

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 hooks/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 info/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 objects/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 refs/

Vous y verrez sans doute d’autres fichiers, mais ceci est un depot qui vient d’etre cree avec git init et c’est ce que vous verrez par defaut. Le fichier description est utilise uniquement par le programme GitWeb, il ne faut donc pas s’en soucier. Le fichier config contient les options de configuration specifiques a votre projet et le repertoire info contient un fichier d’exclusions listant les motifs que vous souhaitez ignorer et que vous ne voulez pas mettre dans un fichier .gitignore. Le repertoire hooks contient les scripts de procedures automatiques cote client ou serveur, ils sont decrits en detail dans Crochets Git.

Il reste quatre elements importants : les fichiers HEAD et (pas encore cree) index, ainsi que les repertoires objects et refs. Ce sont les composants principaux d’un depot Git. Le repertoire objects stocke le contenu de votre base de donnees, le repertoire refs stocke les pointeurs vers les objets *commit* de ces donnees (branches), le fichier HEAD pointe sur la branche qui est en cours dans votre repertoire de travail et le fichier index est l’endroit ou Git stocke les informations sur la zone d’attente. Vous allez maintenant plonger en detail dans chacune de ces sections et voir comment Git fonctionne.

**Les objets de Git**

Git est un systeme de fichier adressables par contenu. Ca veut dire que le coeur de Git est une simple base de paires cle/valeur. Vous pouvez y inserer n’importe quelle sorte de donnees et il vous retournera une cle que vous pourrez utiliser a n’importe quel moment pour recuperer ces donnees. Pour illustrer cela, vous pouvez utiliser la commande de plomberie hash-object, qui prend des donnees, les stocke dans votre repertoire .git, puis retourne la cle sous laquelle les donnees sont stockees

$ find .git/objects

.git/objects

.git/objects/info

.git/objects/pack

$ find .git/objects -type f

Git a initialise le repertoire objects et y a cree les sous-repertoires pack et info, mais ils ne contiennent pas de fichier regulier.

$ echo 'test content' | git hash-object -w –stdin

L’option -w specifie a hash-object de stocker l’objet, sinon la commande repondrait seulement quelle serait la cle. --stdin specifie a la commande de lire le contenu depuis l’entree standard, sinon hash-object s’attend a trouver un chemin vers un fichier. La sortie de la commande est une empreinte de 40 caracteres. C’est l’empreinte SHA-1 ‒ une somme de controle du contenu du fichier que vous stockez plus un en-tete,

Voyez maintenant comment Git a stocke vos donnees :

$ find .git/objects -type f

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

C’est comme cela que Git stocke initialement du contenu ‒ un fichier par contenu, nomme d’apres la somme de controle SHA-1 du contenu et de son en-tete. Le sous-repertoire est nomme d’apres les 2 premiers caracteres de l’empreinte et le

fichier d’apres les 38 caracteres restants.

Vous pouvez recuperer le contenu avec la commande cat-file. Cette commande est un peu le couteau suisse pour l’inspection des objets Git. Lui passer l’option -p ordonne a la commande catfile de determiner le type de contenu et de vous l’afficher joliment :

$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4

D’abord, creez un nouveau fichier et enregistrez son contenu dans la base de donnees:

$ echo 'version 1' > test.txt

$ git hash-object -w test.txt

83baae61804e65cc73a7201a7252750c76066a30

Puis, modifiez le contenu du fichier et enregistrez-le a nouveau :

$ echo 'version 2' > test.txt

$ git hash-object -w test.txt

1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

Votre base de donnees contient les 2 versions du fichier, ainsi que le premier contenu que vous avez stocke ici :

$ find .git/objects -type f

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

Vous pouvez maintenant restaurer le fichier a sa premiere version :

$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt

$ cat test.txt

version 1

Git peut vous donner le type d’objet de n’importe quel objet Git, etant donne sa cle SHA-1, avec cat-file -t :

**Les objets arbres**

Un unique arbre contient une ou

plusieurs entrees, chacune etant l’empreinte SHA-1 d’un blob ou d’un sous-arbre (*sub-tree*) avec ses droits d’acces (*mode*), son type et son nom de fichier associes. L’arbre le plus recent d’un projet pourrait ressembler, par exemple, a ceci :

$ git cat-file -p master^{tree}

100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README

100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile

040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib

La syntaxe master^{tree} signifie l’objet arbre qui est pointe par le dernier *commit* de la branche master

Ces trois objets principaux (le blob, l’arbre et le *commit*) sont initialement stockes dans des fichiers separes du repertoire .git/objects. Voici tous les objets

contenus dans le repertoire exemple, commentes d’apres leur contenu :

On trouve les fichiers contenant des empreintes SHA-1 dans le repertoire git/refs.

$ find .git/refs

.git/refs

.git/refs/heads

.git/refs/tags

$ find .git/refs -type f

C’est simplement ce qu’est une branche dans Git : un simple pointeur ou reference sur le dernier etat d’une suite de travaux. Pour creer une branche a partir du deuxieme *commit*, vous pouvez faire ceci :

**La branche HEAD**

Le fichier HEAD est une reference symbolique a la branche courante. Par reference symbolique, j’entends que contrairement a une reference normale, elle ne contient pas une empreinte SHA-1, mais plutot un pointeur vers une autre reference. Si vous regardez ce fichier, vous devriez voir quelque chose comme ceci :

$ cat .git/HEAD

ref: refs/heads/master

**Etiquettes**

Nous venons de parcourir les trois types d’objets utilises par Git, mais il en existe un quatrieme. L’objet etiquette (*tag* en anglais) ressemble beaucoup a un objet *commit*. Il contient un etiqueteur, une date, un message et un pointeur. La principale difference est que l’etiquette pointe en general vers un *commit* plutot qu’un arbre. C’est comme une reference a une branche, mais elle ne bouge jamais : elle pointe toujours vers le meme *commit*, lui donnant un nom plus sympathique.

Comme presente au Les bases de Git, il existe deux types d’etiquettes : annotee et legere

C’est tout ce qu’est une etiquette legere : une reference qui n’est jamais modifiee.

Une etiquette annotee est plus complexe. Quand on cree une etiquette annotee, Git cree un objet etiquette, puis enregistre une reference qui pointe vers lui plutot que directement vers le *commit*. Vous pouvez voir ceci en creant une etiquette annotee (-a specifie que c’est une etiquette annotee) :

$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'

Voici l’empreinte SHA-1 de l’objet cree :

$ cat .git/refs/tags/v1.1

9585191f37f7b0fb9444f35a9bf50de191beadc2

**References distantes**

Git stocke la valeur que vous avez poussee en dernier vers cette reference pour chaque branche dans le repertoire refs/remotes

Ensuite, vous pouvez voir l’etat de la branche master dans la reference distante origin la derniere fois que vous avez communique avec le serveur en regardant le fichier refs/remotes/origin/master :

$ cat .git/refs/remotes/origin/master

ca82a6dff817ec66f44342007202690a93763949

**Github**

git ls-remote origin affcihent la liste de toutes les branches et etiquettes ainsi que d’autres references dans le depot.

**Git distribue**

lancez donc git log --no-merges dessus pour voir a quoi ressemble un historique

de *commits* avec des messages bien formates.

La difference principale reste que les fusions ont lieu du cote client plutot que

sur le serveur au moment de valider

Il est tres important de comprendre ceci si vous avez deja utilise Subversion, parce qu’il faut remarquer que les deux developpeurs n’ont pas modifie le meme fichier. Quand des fichiers differents ont été modifies, Subversion realise cette fusion automatiquement sur le serveur alors que Git necessite une fusion des modifications locale. John doit recuperer les modifications de Jessica et les fusionner avant d’etre autorise a pousser :

$ git log --no-merges issue54..origin/master

La syntaxe issue54..origin/master est un filtre du journal qui ordonne a Git de ne montrer que la liste des *commits* qui sont sur la seconde branche (dans ce cas origin/master) et qui ne sont pas sur la premiere (dans ce cas prob54). Nous aborderons cette syntaxe en detail dans Plages de *commits.*

Mais il y a un petit probleme — elle doit pousser son travail fusionne dans sa branche fonctionB sur la branche fonctionBee du serveur. Elle peut le faire en specifiant la branche locale suivie de deux points (:) suivi de la branche distante a la commande git push :

$ git push -u origin fonctionB:fonctionBee

Notez l’option -u. C’est un raccourci pour --set-upstream, qui configure les branches pour faciliter les poussees et les tirages plus tard.

Vous demarrez une nouvelle branche a partir de la branche origin/master

courante, y collez les modifications de fonctionB en resolvant les conflits, changez l’implementation et poussez le tout en tant que nouvelle branche

$ git checkout -b fonctionBv2 origin/master

$ git merge --no-commit --squash fonctionB

# (changement d'implementation)

$ git commit

$ git push macopie fonctionBv2